

# Ranked Enumeration for Database Queries

Nikolaos Tziavelis<sup>\*</sup>  
UC Santa Cruz  
ntziavel@ucsc.edu

Wolfgang Gatterbauer  
Northeastern University  
w.gatterbauer@northeastern.edu

Mirek Riedewald  
Northeastern University  
m.riedewald@northeastern.edu

## ABSTRACT

Ranked enumeration is a query-answering paradigm where the query answers are returned incrementally in order of importance (instead of returning all answers at once). Importance is defined by a ranking function that can be specific to the application, but typically involves either a lexicographic order (e.g., “ORDER BY R.A, S.B” in SQL) or a weighted sum of attributes (e.g., “ORDER BY 3\*R.A + 2\*S.B”). Recent work has introduced *any-k algorithms* for (multi-way) join queries, which push ranking into joins and avoid materializing intermediate results until necessary. The top-ranked answers are returned asymptotically faster than the common join-then-rank approach of database systems, resulting in orders-of-magnitude speedup in practice.

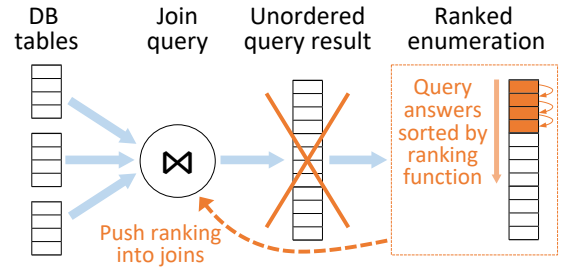
In addition to their practical usefulness, these techniques complement a long line of theoretical research on *unranked enumeration*, where answers are also returned incrementally, but with no explicit ordering requirement. For a broad class of ranking functions with certain monotonicity properties, including lexicographic orders and sum-based rankings, the ordering requirement surprisingly does not increase the asymptotic time or space complexity, apart from logarithmic factors.

A key insight is the connection between ranked enumeration for database queries and the fundamental task of computing the  $k^{\text{th}}$ -shortest path in a graph. Although this connection is important for grounding the problem in the literature, it can obfuscate the simplicity of the algorithm. In this article, we adopt a pragmatic approach and present a slightly simplified version of the algorithm without the shortest-path interpretation. We believe that this will benefit practitioners looking to implement and optimize any- $k$  approaches.

## 1 Introduction

Data analytics queries can generate large intermediate or final results, rendering data systems unresponsive. A primary culprit is the join operator,

<sup>\*</sup>Work done while at Northeastern University.



**Figure 1:** Enumerating the query answers in ranked order without first materializing the unordered query result. *Sorting is pushed into the join operation* so that joining and ranking are interleaved.

which combines data from different tables, potentially causing a combinatorial explosion in the output. Consequently, traditional join-processing techniques can become infeasible, or they simply take too long before delivering any answer to the user or to the next step in a data-processing pipeline. Work on *enumeration* [6, 41] addresses this by returning query answers incrementally as quickly as possible, even when the full query output is too large to compute. However, enumeration traditionally does not support a desired order (or *ranking*) specifying, which answers should be returned first. We thus refer to it as *unranked enumeration*. In practice, certain answers may be preferred over others based on some notion of importance or relevance. For instance, higher importance may be assigned to newer or more trusted data. *Ranked enumeration* [24, 44] therefore augments enumeration with a total-order feature over the query answers, formalized by a ranking function (e.g., expressed by an ORDER BY clause in SQL).

Database systems today follow a join-then-rank approach, i.e. they first compute *all join answers* and then apply the ranking (by sorting either incrementally or in batch). One way to think about the improvement we seek is that we want to “push”

```

SELECT Cit1.PaperID, Cit2.PaperID, Cit3.PaperID,
       Cit3.CitedPaperID, Cit1.InflWeight +
       Cit2.InflWeight + Cit3.InflWeight AS Weight
FROM Cit Cit1, Cit Cit2, Cit Cit3
WHERE Cit1.CitedPaperID = Cit2.PaperID AND
      Cit2.CitedPaperID = Cit3.PaperID
ORDER BY Weight

```

**Figure 2:** SQL query for ranking chains of highly influential citations.

the ranking operator deeper into the query plan. While this resembles typical database optimizations, such as pushing projections before joins, the task is more challenging, because join and ranking operators generally do not commute. Novel algorithms are required, where joining and ranking are interleaved.<sup>1</sup>

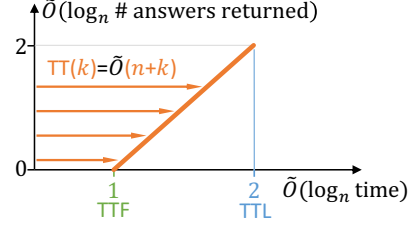
**Performance Goal.** How can performance for such an algorithm be measured? The top-ranked answers should be returned quickly without wasting resources on low-ranked ones, similar to classic top- $k$  queries [29]. However, in contrast to top- $k$ , where “pruning” techniques based on the *given* number of returned answers  $k$  can be leveraged,<sup>2</sup> a ranked-enumeration algorithm does not know the value  $k$  in advance. Instead of pruning, it can at best *postpone* work on lower-ranked answers, providing guarantees *no matter how many answers are eventually returned*. We are thus interested in the *Time-To- $k$* , or  $TT(k)$ , for any possible value of  $k$ . This gave rise to the “any- $k$ ” label, quasi an “any-time top- $k$ ” algorithm [14, 53, 54].

Note that a stricter and popular [6, 28, 38, 41] measure of performance involves combining *preprocessing time* (i.e.,  $TT(1)$ ) with the *worst-case delay between answers* (i.e., the maximum inter-arrival time). However, lowering the worst-case delay may have no practical benefit if it does not also improve  $TT(k)$  [18, 20, 46]. Adopting  $TT(k)$  allows for situations where a spike in delay is offset by shorter delays in *previous* iterations. An established example where this difference occurs is incremental Quick-Sort [40] which guarantees  $TT(k) = \mathcal{O}(n + k \log k)$ , but has a linear worst-case delay between answers.

**An Example.** Consider a bibliography dataset that stores the influence of research papers on later

<sup>1</sup>Even simpler top-1 queries are not efficiently supported by current systems. For a minimum example in PostgreSQL, see slide 20: [https://northeastern-datalab.github.io/cs7240/sp24/download/cs7240-T3-U1-Acyclic\\_Queries.pdf](https://northeastern-datalab.github.io/cs7240/sp24/download/cs7240-T3-U1-Acyclic_Queries.pdf).

<sup>2</sup>Besides the requirement of  $k$  being fixed in advance, older work on top- $k$  joins assumes a cost model that accounts for data access, but not for intermediate results [48, Part 1].



**Figure 3:** Ranked enumeration guarantees for the query of Figure 2: The first answer (TTF for Time-To-First) is returned in  $\tilde{\mathcal{O}}(n)$  and the last answer (TTL for Time-To-Last) in  $\tilde{\mathcal{O}}(n^2)$ .

papers that cite them. Each tuple in relation  $\text{Cit}(\text{PaperID}, \text{CitedPaperID}, \text{InflWeight})$  states that a paper with ID  $\text{CitedPaperID}$  influenced a later paper  $\text{PaperID}$  with a numerical weight  $\text{InflWeight}$ . For the sake of the example, assume that the influence weight has been precomputed by some prediction technique and takes on an integer value in range  $[1, 10]$ , with 1 being the most influential. To extract chains of highly influential citations, we can write the join query

$$\text{Cit}(p_1, p_2, s_1), \text{Cit}(p_2, p_3, s_2), \text{Cit}(p_3, p_4, s_3)$$

and order its answers in ascending sequence of the SUM  $s_1 + s_2 + s_3$ . For readers unfamiliar with Datalog, note that relation  $\text{Cit}$  appears three times to indicate a self-join (which requires renaming to  $\text{Cit1}, \text{Cit2}, \text{Cit3}$  in SQL as shown in Figure 2) and that a variable like  $p_2$  appearing more than once indicates an equi-join between the corresponding columns (i.e.,  $\text{Cit1}.p_2 = \text{Cit2}.p_2$ ). How fast can ranked enumeration be here? The entire query output can have size  $n^2$  in the worst case [5]. On the other hand, simply checking if any query answer exists (called the *Boolean* query) takes  $\Theta(n)$  [55]. Ranked enumeration aims to cover the continuum between those two with  $TT(k) = \tilde{\mathcal{O}}(n+k)$ , as shown in Figure 3. The  $\tilde{\mathcal{O}}$  notation abstracts away logarithmic factors in  $n$  and  $k$  introduced by join indexes or sorting (by  $s_1 + s_2 + s_3$ ).

**Prioritizing Computation.** To build intuition, let us first consider how unranked enumeration works. If we were to follow a standard table-at-a-time approach, we would start by joining  $\text{Cit1} \bowtie \text{Cit2}$ . This is a costly bulk computation of time complexity  $\tilde{\mathcal{O}}(n^2)$ . However, it would not yet produce a single query answer because table  $\text{Cit3}$  has not been checked. To produce answers as quickly as possible, we need to be more careful in where we spend resources and prioritize differently. Instead of a table-at-a-time, a *tuple-at-a-time* approach is needed. We start with only one tuple from  $\text{Cit1}$ , look up the

matches in Cit2, pick one, and then look up the Cit3 matches to produce one answer. This strategy can be implemented using a pipelined execution in a database system. The standard unranked enumeration algorithm [6] achieves  $\tilde{O}(n + k)$  by following such an approach, preceded by a  $\tilde{O}(n)$ -time semi-join reduction [55], which removes “dangling” tuples that do not contribute to the final output.

Ranked enumeration appears more challenging because additional prioritization is required to avoid low-ranking query answers. Interestingly, a more careful look at the unranked enumeration algorithm [6] reveals that, with appropriate sorting of the input relations, the output naturally follows a *lexicographic order*. A lexicographic order is defined by a sequence of variables, such as  $s_1 \rightarrow s_2 \rightarrow s_3$ . It means that the answers are first ordered by variable  $s_1$ , then by  $s_2$ , and then by  $s_3$  (ORDER BY Cit1.InflWeight, Cit2.InflWeight, Cit3.InflWeight in SQL). This heavily prioritizes the weight of the first citation in the chain; a chain with weights  $1 \rightarrow 10 \rightarrow 10$  would be ranked higher than a chain with weights  $2 \rightarrow 1 \rightarrow 1$ . The enumeration algorithm of Bagan et al. [6] is capable of producing such an order, granted that we first sort each copy of Citi by InflWeight.

But what if a different order that is “inconvenient” for the algorithm is required? As we will discuss in more detail, certain lexicographic orders, such as  $s_2 \rightarrow s_1 \rightarrow s_3$  cannot be achieved by this approach. Moreover, for SUM ranking, the situation is more difficult because a high-ranking tuple in Cit1 might only join with low-ranking tuples in Cit2 and Cit3, leading to low-ranking answers in aggregate. Addressing this requires a stronger form of prioritization that incorporates *lookahead* information about tuples and weights that come later in the query plan.

**Any- $k$  Algorithms.** Recent developments led to the design and implementation of any- $k$  algorithms achieving  $TT(k) = \tilde{O}(n + k)$  for acyclic join queries and appropriately monotone ranking functions [24, 44]. These include *all* lexicographic orders, SUM, as well as MIN and MAX. In our example, the first  $k = \mathcal{O}(1)$  answers are obtained after only  $\tilde{O}(n)$ , and—if the enumeration is carried out to the end—the last answer in  $\tilde{O}(n^2)$ , matching the join-then-rank approach. Compared to unranked enumeration, ranking by  $s_1 + s_2 + s_3$  introduces only a logarithmic factor in  $k$ .

Although multiple any- $k$  algorithms exist, their complexity differences concern logarithmic factors and treating query size as a variable that can grow arbitrarily, which may not always materialize in practice. In this article, we cater to practicality and

ease of understanding, focusing on data complexity, guarantees in  $\tilde{O}$  without logarithmic factors, and on the easiest-to-understand variant.<sup>3</sup> We describe the algorithm in a streamlined way, without the graph abstraction that has been used [44] to highlight the connection to earlier work on shortest-path enumeration [30, 33].

**Organization.** The rest of this article is organized as follows. Section 2 introduces necessary concepts and notation. Section 3 presents a simple algorithm that works for certain lexicographic orders and explores which lexicographic orders are achievable with this algorithm. Section 4 takes on the harder case of SUM. Section 5 discusses several extensions that generalize the approach to more expressive queries and ranking functions. Section 6 concludes and provides directions for future work.

## 2 Basic Concepts

We focus on Select-Project-Join queries, which we formally define in the usual way as Conjunctive Queries. Throughout the article, we use  $[m]$  to denote the set of integers  $\{1, \dots, m\}$ .

**Database.** A *database*  $D$  is a set of finite relations  $\{R_1, \dots, R_m\}$ , where each  $R_i$  for  $i \in [m]$  has arity  $\text{ar}(R)$  (i.e.,  $\text{ar}(R)$  attributes or columns) and draws values from a fixed infinite domain  $\text{dom}$ , (i.e.,  $R_i \subseteq \text{dom}^{\text{ar}(R_i)}$ ). The size of the database  $n$  is the number of tuples across all relations.

**Query.** In Datalog, a *Conjunctive Query* (CQ)  $Q$  is an expression  $Q(\mathbf{Y}) :- R_1(\mathbf{V}_1), \dots, R_\ell(\mathbf{V}_\ell)$ , where each  $\mathbf{V}_i$  for  $i \in [\ell]$  is a list of either *variables* (representing database attributes) or constants from  $\text{dom}$  (encoding selection). Each *atom*  $R_i(\mathbf{V}_i)$  refers to a (not necessarily distinct) database relation with  $|\mathbf{V}_i|$  attributes. If  $\mathbf{X}$  is the set of all distinct variables appearing in all lists  $\mathbf{V}_i$  for  $i \in [\ell]$ , then the variables  $\mathbf{Y}$  (representing output attributes) need to be a subset of  $\mathbf{X}$  and are called *free*. A *Join Query* (JQ) is a special case of a CQ where all variables are free (i.e.,  $\mathbf{Y} = \mathbf{X}$ ). Multiple atoms are allowed to refer to the same relation, resulting in a *self-join*. The query size, measured by the number of symbols in the query, is assumed to be  $\mathcal{O}(1)$ . This is often referred to as *data complexity* [51] and it is relevant in practice because while new data may be collected, the query size does not typically grow unboundedly.

Queries are evaluated over a database  $D$  and produce a result  $Q(D)$ . A *query answer* or *output tuple* is an element  $q \in Q(D)$ . The occurrence of the same

<sup>3</sup>The specific variant we present is ANYK-PART with eager sorting [44, Figure 6].

variable in different atoms encodes an *equi-join* condition, implying equality between the corresponding attributes. A typical preprocessing step for all algorithms is to (1) remove self-joins from the query by copying database tables and (2) remove selections on individual relations (like  $R(x, 1)$  or  $R(x, x)$ ) by filtering. These operations take  $\mathcal{O}(n)$  and can be ignored because the cost is asymptotically the same as reading the database once. Afterwards, a naive evaluation strategy to compute  $Q(D)$  (which helps to understand the query semantics) is to (i) materialize the Cartesian product of the  $\ell$  relations, (ii) select tuples that satisfy the equi-joins, and (iii) project on the  $\mathbf{Y}$  attributes.

**Acyclicity.** A CQ is (alpha-)acyclic [16] if it admits a *join tree*. A join tree is a rooted tree whose nodes are the query atoms and for each variable  $x$ , all tree nodes containing  $x$  form a connected subtree.<sup>4</sup> The acyclicity of a CQ can be tested, and a corresponding join tree can be constructed, in linear time in the query size [42].

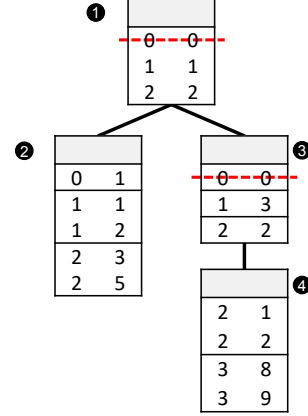
**Ranking.** Ranked enumeration assumes a user-specified *ranking function* that orders the query answers  $Q(D)$  by mapping them to a domain  $W$  equipped with a total order  $\preceq$ . Ties are broken arbitrarily. Given a query  $Q$ , a *lexicographic order*  $L$  is a sequence of query variables  $x_1 \rightarrow x_2 \rightarrow \dots$ , implying that the answers are first compared by the values of  $x_1$ , and if tied by the values of  $x_2$ , and so on. A *partial* lexicographic order contains a strict subset of the query variables. Another case is *SUM*, given by an expression  $f_1(x_1) + f_2(x_2) + \dots$ , where  $f_1, f_2, \dots$  can be arbitrary,  $\mathcal{O}(1)$ -computable functions mapping  $\text{dom}$  to  $\mathbb{R}$ . The ranking may alternatively be defined using values on the database tuples instead of the query variables; the latter can be reduced to the former in linear time as we will see in more detail in Section 4.1.

### 3 Enumeration by Lexicographic Order

We begin with the lexicographic orders that can be produced as a by-product of the standard “un-ranked” enumeration algorithm through a minor extension (i.e., pre-sorting all input relations according to the lexicographic order). Although various descriptions of this algorithm exist in the literature using different abstractions [6, 11, 38, 41], it is often overlooked that it can easily produce query answers according to certain lexicographic orders.

We offer a detailed description that (i) is easy to implement and (ii) generalizes to SUM (Section 4)

<sup>4</sup>For an illustration, please see <https://www.youtube.com/watch?v=toi7ysuyRkw&t=340> [49].



**Figure 4:** An example database for the join query  $R(x_1, x_2), S(x_1, x_3), T(x_2, x_4), U(x_4, x_5)$ . The relations are organized in a join tree. Red marks indicate tuples removed by the semijoin reduction. Also shown are shared variables between child-parent pairs and the relation ordering **rel** used by the lexicographic enumeration algorithm.

and other orders. We focus on acyclic JQs and discuss how this restriction can be lifted in Section 5.

The algorithm consists of two phases. First, the preprocessing phase builds essential data structures such as join indexes and applies a semijoin reduction [55] to remove dangling tuples from the input relations. Then, the enumeration phase traverses the relations using the indexes to connect joining tuples. The  $\tilde{\mathcal{O}}(n+k)$  complexity guarantee for  $\text{TT}(k)$  hinges on the semijoin filtering, which eliminates “dead-ends” by ensuring that every partial query answer—generated by joining tuples from a subset of relations—can be extended to a complete query answer. We will detail both phases in Sections 3.1 and 3.2, then examine, which lexicographic orders can be supported by this algorithm in Section 3.3.

As a guiding example, we use the query  $R(x_1, x_2), S(x_1, x_3), T(x_2, x_4), U(x_4, x_5)$  and show how to achieve the order  $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4 \rightarrow x_5$ . An example database is shown in Figure 4.

#### 3.1 Bottom-up Preprocessing Phase

**Join Order.** The preprocessing phase starts by organizing the relations in a (rooted) join tree  $\mathcal{T}$ . Unlike a database query plan, a join tree does not fully specify the join order. It only determines that a parent relation must be processed before its children (also called a topological sort). Hence any order that respects this constraint can be followed by the enumeration algorithm. Let function **rel** denote such an order, i.e., it maps the integers  $[\ell]$  to database relations, where  $\ell$  is the number of rela-



tions. In our example, we have  $\text{rel}(1) = R, \text{rel}(2) = S, \text{rel}(3) = T, \text{rel}(4) = U$  (see Fig. 4). This means that relation  $S$  will always be visited before relation  $T$  during enumeration. To encode the tree structure, we refer to the parent of the  $r$ -th relation in the order as  $\text{pr}(r)$ , for  $r \in [2, \ell]$ .

**Join Indexes.** Next, we build join indexes, e.g., B-trees or hash indexes, allowing us to find matching tuples efficiently. We abstract an index as a function  $\text{JoinIndex}_{R \rightarrow S}$ , which, given a tuple  $t \in R$ , returns a list  $M$  of  $S$  tuples that agree with  $t$  on the join attributes between  $R$  and  $S$  (i.e., the common variables between the atoms). To achieve the desired  $\text{TT}(k)$  guarantees, the index must be built in  $\tilde{O}(n)$  with lookups in  $\tilde{O}(1)$  (not including the time it takes to read  $M$ ). We construct one index for each parent-child pair in the join tree, i.e.,  $\text{JoinIndex}_{R \rightarrow S}$  based on  $x_1$ ,  $\text{JoinIndex}_{R \rightarrow T}$  based on  $x_2$ ,  $\text{JoinIndex}_{T \rightarrow U}$  based on  $x_4$  in our example. Figure 4 shows each relation grouped by the attributes that join with the parent, i.e., the image of  $\text{JoinIndex}_{\text{pr}(r) \rightarrow \text{rel}(r)}$  for the  $r$ -th relation,  $r \in [2, \ell]$ . The root has no grouping.

**Semijoin reduction.** Using the join tree and indexes, we perform a *semijoin reduction* exactly as in the bottom-up step of the Yannakakis algorithm [55]. The relations are traversed in reverse topological order with a semijoin applied for each parent-child pair. In our example, the semijoins are executed in the following order:

$$T = T \ltimes U, \quad R = R \ltimes T, \quad R = R \ltimes S$$

This step is crucial for our desired complexity guarantee. To understand why, consider tuple  $R(0, 0)$ , for which there are matching tuples in  $S$  and  $T$ , but none in  $U$ . Consequently, the time processing  $R(0, 0)$  is wasted, without producing an output tuple. With sufficiently many such “dangling” tuples, the time between consecutive answers would grow to exceed  $\text{TT}(k) = \tilde{O}(n + k)$ . The semijoin reduction prevents this by removing dangling tuples like  $R(0, 0)$  and  $T(0, 0)$ . Notice that  $S(0, 1)$  is dangling, but not removed. Removing all dangling tuples would require a *full reduction* [9, 13], which is not necessary for the enumeration algorithm. It is easy to show that any remaining dangling tuples will never be accessed by top-down traversals.

Algorithm 1 presents the semijoin reduction expressed in a way that easily generalizes to support other orders, as we will see in Section 4. Specifically, it can be viewed as message passing at the tuple level: Each tuple pulls “messages” from joining tuples in the children relations, determines its own state based on the messages, and later passes a mes-

---

**Algorithm 1:** Preprocessing for lexicographic enumeration (Section 3.1).

---

**Input:** acyclic JQ  $Q$  (without self-joins), database  $D$ , join tree  $\mathcal{T}$ , lexicographic order  $L$ , relation ordering  $\text{rel}$  consistent with  $\mathcal{T}$

```

1 Output: reduced and sorted database  $D'$ ,  $\text{JoinIndex}_{R \rightarrow S}$ 
   for each parent  $R$  and child  $S$  in  $\mathcal{T}$ 
2 Initialize  $\text{val}(t) = \text{True}$  for all tuples  $t$  of all relations
3 //Process relations in reverse  $\text{rel}$  order (bottom-up in  $\mathcal{T}$ )
4 for  $i = \ell$  down to 2 do
5   relation  $S = \text{rel}[i]$ ; relation  $R = \text{pr}(S)$ 
6   //Relation  $S$  has been reduced in a previous iteration
7   (or is a leaf)
8   Construct  $\text{JoinIndex}_{R \rightarrow S}$  on shared attributes
9   Sort  $\text{JoinIndex}_{R \rightarrow S}$  entries by  $L$ 
10  for tuple  $t \in R$  do
11     $M = \text{JoinIndex}_{R \rightarrow S}(t)$ 
12    //Memoization:  $\text{val}(M)$  is reused
13    if  $\text{val}(M)$  not already computed then
14       $\text{val}(M) = \text{False} \vee \bigvee_{t' \in M} \text{val}(t')$ 
15       $\text{val}(t) = \text{val}(t) \wedge \text{val}(M)$ 
16      if not  $\text{val}(t)$  then remove  $t$  from  $R$  in  $D$ 
17 Sort the root  $R$  by  $L$ 
18 return  $D, \text{JoinIndex}_{R \rightarrow S}$  for all  $(R, S) \in \mathcal{T}$ 

```

---

sage up the tree. The “message” here is a Boolean value that indicates whether matching tuples exist in the subtree. If the aggregated message from at least one of the children relations is “False”, then the tuple is removed and a “False” message is propagated upwards. Note that parents are “pulling” instead of children “pushing” messages so that we can use the parent-to-child join indexes that we anyway need in the enumeration phase. The algorithm employs *memoization* for the aggregated message of a join group (Line 13), since multiple tuples in the parent relation may access it. This is important in order to guarantee linear time.

**Sorting.** When we build a join index, we sort its entries (i.e., the tuples within the same join group) by the same lexicographic order. In the example, the entries of  $\text{JoinIndex}_{R \rightarrow S}$  are sorted by  $x_1 \rightarrow x_3$ , the entries of  $\text{JoinIndex}_{R \rightarrow T}$  by  $x_2 \rightarrow x_4$ , and so on. The join index is built after reducing a relation with messages from its children, and sorted thereafter. The tuples of the root relation are considered to belong to the same join group (as if a parent relation with an empty set of join variables to group-by existed) and are also sorted. Slightly abusing the notation, we treat a relation as a sorted list of tuples; e.g.,  $R[1]$  denotes the first tuple of  $R$ .

### 3.2 Top-down Enumeration Phase

While the semijoin reduction proceeds bottom-up in the opposite direction of relation order  $\text{rel}$  (Line 4), the enumeration phase traverses the relations top-down. We start with tuple  $R[1] = R(1, 1)$  and, through the join indexes, find the first match

in every relation, yielding the first query answer  $(1, 1, 1, 3, 8)$ .<sup>5</sup> In the second iteration, we proceed with the matches from the last relation, i.e., tuple  $(3, 9)$  from  $U$ , obtaining  $(1, 1, 1, 3, 9)$ . This exhausts all matches in  $U$ , therefore in the third iteration the algorithm backtracks to the next match in preceding relation  $T$ . Since no second match exists in  $T$ , we backtrack once again to  $S$ , encountering  $(1, 2)$  there. With  $(1, 1, 2)$  as a partial answer, the algorithm proceeds forward to  $T, U$  to obtain  $(1, 1, 2, 3, 8)$ . The process continues analogously, returning answers  $(1, 1, 2, 3, 9)$ ,  $(2, 2, 3, 2, 1)$ , etc.<sup>6</sup>

This enumeration can be implemented recursively, akin to a standard depth-first search (DFS). Equivalently, we implement it with a *stack* of partial query answers (LIFO), which tracks the current frontier. A partial answer contains matched tuples from only a subset of the relations. When a partial answer is popped from the stack and we extend it into a complete answer, alternatives that use the next available tuple are pushed back onto the stack, starting from the current relation. The extension of a partial answer always selects the first matching tuple following the relation order. This is illustrated in Figure 5. Notice that when the second answer  $(1, 1, 1, 3, 9)$  is popped, the current relation is  $\text{rel}(4) = U$ , with  $R(1, 1)$ ,  $S(1, 1)$ , and  $T(1, 3)$  from the earlier relations considered fixed (so we do not generate new answers from those relations). In fact,  $(1, 1, 2)$  is already on the stack from the previous iteration. This logic ensures that we enumerate each query answer exactly once. For the time complexity, note that we visit each relation at most once in each iteration, thus the cost per iteration is  $\tilde{O}(1)$  because query size is treated as a constant.

The LIFO nature of the stack is essential for achieving the lexicographic order. For instance, new answers that replace  $U$ -tuples (thus, change only the  $x_5$  value) are always popped before answers that replace tuples in  $R$ ,  $S$ , or  $T$ . In the following, we discuss the achievable orders in more detail.

### 3.3 Supported Lexicographic Orders

Different lexicographic orders can be achieved by different sortings of the individual relations. For example, if we sort  $R$  by  $x_2 \rightarrow x_1$ , we can achieve the order  $x_2 \rightarrow x_1 \rightarrow x_3 \rightarrow x_4 \rightarrow x_5$  without any other change in the algorithm. Some other orders can be achieved

<sup>5</sup>Answers are represented as a tuple of values assigned to variables  $(x_1, \dots, x_5)$ , or alternatively, as a list of joining tuples  $[t_1, \dots, t_4]$ . For ease of presentation, we use the former in text and the latter in pseudocode.

<sup>6</sup>For an illustration, please see <https://www.youtube.com/watch?v=toi7ysuyRkw&t=1720s> [49].

---

#### Algorithm 2: Ranked enumeration for lexicographic orders without disruptive trios

---

**Input:** acyclic JQ  $Q$ , database  $D$ , lexicographic order  $L$  without disruptive trio

**Output:** Ranked enumeration of  $Q(D)$  in  $L$  order

```

1 Remove self-joins from  $Q$  by copying the corresponding
  relations and renaming them in both  $D$  and  $Q$ 
2 Construct an  $L$ -consistent join tree  $\mathcal{T}$  of  $Q$  with
   $L$ -consistent relation order given by  $\text{rel}(i), i \in [\ell]$ 
3 Preprocess( $Q, D, \mathcal{T}, L, \text{rel}$ ) (Algorithm 1)
4 //A partial answer in the stack is represented as a list of
  input tuples together with their positions in the
  corresponding join groups (to easily get the next)
5 Initialize stack  $\mathcal{S}$  with element  $[(t_1, 1)]$  where  $t_1 = R[1]$  and
   $R$  is the root of  $\mathcal{T}$ 
6 repeat
7   //Pop a partial answer ( $1 \leq r \leq |\mathcal{T}|$ ), which also
   contains the positions  $j_i$  for each tuple  $t_i$ 
8    $s = \mathcal{S}.\text{pop}()$ ;  $[(t_1, j_1), \dots, (t_r, j_r)] = s$ 
9   //Look up matches in  $r$ -th relation
10   $M_r = \text{JoinIndex}_{\text{pr}(r) \rightarrow \text{rel}(r)}(t_{\text{pr}(r)})$ 
11  //Push partial answer with next tuple of  $r$ -th relation.
   It exists if  $j_r$  is not the last position in the group  $M_r$ 
12  if  $|M_r| \geq j_r + 1$  then
13     $s' = s.\text{copy}().\text{replaceLast}((M_r[j_r + 1], j_r + 1))$ 
14     $\mathcal{S}.\text{push}(s')$ 
15  //Range over the remaining relations
16  for  $i$  from  $r + 1$  to  $|\mathcal{T}|$  do
17    //Look up matches in  $i$ -th relation
18     $M_i = \text{JoinIndex}_{\text{pr}(i) \rightarrow \text{rel}(i)}(t_{\text{pr}(i)})$ 
19    //Extend partial answer with first tuple in matches
    of  $i$ -th relation
20     $t_i = M_i[1]$ ;  $s.\text{append}((t_i, 1))$ 
21    //s is now  $[(t_1, j_1), \dots, (t_r, j_r), \dots, (t_i, 1)]$ 
22    if  $|M_i| \geq 2$  then
23      //Push partial answer with next tuple of  $i$ -th
      relation to stack  $\mathcal{S}$ 
24       $s' = s.\text{copy}().\text{replaceLast}((M_i[2], 2))$ 
25       $\mathcal{S}.\text{push}(s')$ 
26  Merge  $s$  into single tuple and output
27 until query is interrupted or  $\mathcal{S}$  is empty

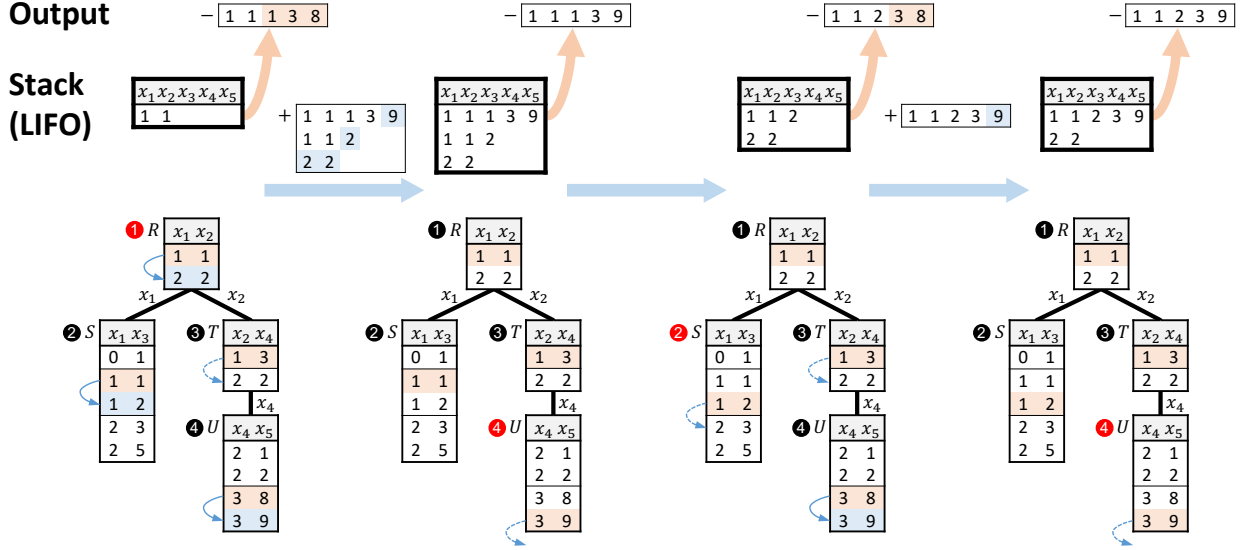
```

---

by additionally selecting a different topological sort on the join tree. With  $[R, T, U, S]$  instead of  $[R, S, T, U]$ , we can achieve  $x_1 \rightarrow x_2 \rightarrow x_4 \rightarrow x_5 \rightarrow x_3$ . However, certain lexicographic orders cannot be achieved by this algorithm. Brault-Baron [15] identified a sufficient condition, which was later termed a *disruptive trio* [21] and shown to be necessary for other problems related to enumeration. (We discuss this in more detail in Section 5.4.)

**DEFINITION 1 (DISRUPTIVE TRIO).** For a CQ  $Q$  and lexicographic order  $L$ , three variables  $x_1, x_2, x_3$  from  $L$  with relative order  $x_1 \rightarrow x_2 \rightarrow x_3$  form a disruptive trio if  $x_1$  and  $x_2$  are not neighbors (i.e., they do not appear together in a  $Q$  atom), but  $x_3$  is a neighbor of both  $x_1$  and  $x_2$ .

In our example,  $x_1, x_4, x_2$  form a disruptive trio if  $L$  is  $x_1 \rightarrow x_4 \rightarrow x_2$  or even  $x_1 \rightarrow x_3 \rightarrow x_4 \rightarrow x_5 \rightarrow x_2$ . Intuitively, during the enumeration, we cannot transition from  $R$  to  $T$  without fixing  $x_1, x_2$  before  $x_4$ ,



**Figure 5:** Enumeration steps for the first 4 answers by Algorithm 2 described in Section 3. The stack, shown on top, pops a partial answer, which is extended with the first matching tuples (in orange color) and moved to the output in each iteration. Starting from the last relation for which a tuple is in the partial answer (in red color), we check if a “next” tuple in the same join group exists (in blue color) and push a new answer to the stack. Dashed arrows indicate that there is no next.

which is inconsistent with the order  $L$ .

Braut-Baron showed that for lexicographic orders containing all free variables, the absence of a disruptive trio is equivalent to  $L$  being a *reverse alpha elimination order* [15, Theorem 15], and for partial lexicographic orders, it is equivalent to the lexicographic order being *consistent* with (or, in other words, a restriction of) a reverse alpha elimination order. An alpha elimination order is an ordering of the variables that guides the join tree construction [16]. If variable  $y$  follows variable  $x$  in the elimination order, then in the resulting join tree,  $y$  will never appear without  $x$  in any ancestor of a node that contains  $x$ .<sup>7</sup> This guarantees that there exists a relation ordering  $\text{rel}$  such that the  $L$  variables are encountered in the desired sequence. We call such an ordering of the relations, and its corresponding join tree,  *$L$ -consistent*.

If a desired lexicographic order has no disruptive trio, then we can find an  $L$ -consistent join tree and an  $L$ -consistent ordering of the relations to use with the enumeration algorithm discussed above.

**THEOREM 2 (LEX).** *Let  $Q$  be an acyclic join query over database  $D$  and  $L$  a lexicographic order of the variables in  $Q$ . If  $L$  does not contain a disruptive trio, then ranked enumeration of  $Q(D)$  by*

<sup>7</sup>A similar property has been proposed in factorized databases in order to detect whether a lexicographic order is admissible with a given factorization order [7].

$L$  can be achieved with  $\text{TT}(k) = \tilde{O}(n + k)$ .

Algorithm 2 shows the pseudocode. After the preprocessing phase, a loop returns query answers iteratively by popping and pushing from the stack. Notice that, for each answer, the algorithm keeps track of the positions  $j_1, \dots, j_\ell$  of the tuples within the corresponding join group. This allows it to quickly access the next tuple in the group when constructing new answers (in Lines 13 and 24).

What about the lexicographic orders that contain disruptive trios? Algorithm 2 does not apply because there is no join tree that can match the order. For these orders, as well as SUM, we need a different strategy. Lexicographic orders with disruptive trios can in fact be reduced to a SUM-ordering problem by assigning the appropriate variable weights: If all relations have cardinality at most  $n$ , we can achieve that by setting the weight of the  $i^{\text{th}}$  value of the  $j^{\text{th}}$  variable in the order to  $i \cdot n^{|L|-1-j}$ .

## 4 Enumeration by SUM Order

In this section, we shift focus to ranking by SUM. Let  $\sum_{i=1}^5 x_i$  (in ascending order) be the ranking function for our example query. A naive strategy is to select the best tuple from each relation based on its individual weight. For instance, using the same join tree as before, we could start with  $R(1, 1)$  since it has the lowest weight  $1 + 1 = 2$  within  $R$ . However, this strategy is not guaranteed to find the

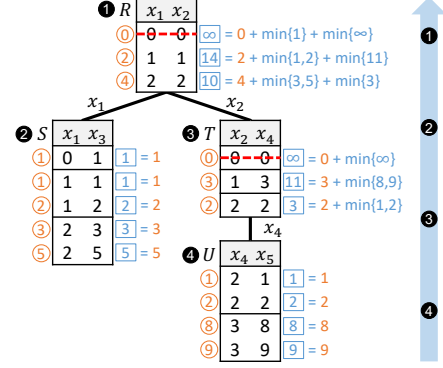
top answers, at least not within the time bounds we aim for. Once we choose  $R(1, 1)$ , we will be stuck in a region of query answers with high overall weight because of the high weights of  $U(3, 8)$  and  $U(3, 9)$ , which are the only matching tuples in  $U$ . The true top-1 answer  $(2, 2, 3, 2, 1)$  starts with  $R(2, 2)$ , which matches with  $U(2, 1)$ . To make the right choices in  $R$ , the algorithm needs “lookahead” information about later matches in relations like  $U$ .

Unfortunately, it is infeasible to explicitly precompute the “lookahead” combinations of  $S, T, U$  tuples in the preprocessing phase, because that would exceed our desired  $\tilde{O}(n)$ . Instead, we rely on Dynamic Programming and a factorized representation of the query output. The enumeration phase is similar to the algorithm of Section 3.2, but uses a *priority queue* instead of a *stack* in order to prioritize the candidates according to the “lookahead” information computed during preprocessing.

#### 4.1 Bottom-up Preprocessing Phase

To prioritize the tuples that lead to the lowest total weight, we modify the semijoin reduction so that, in addition to removing dangling tuples, we also compute the *best possible weight*  $\text{opt}(t)$  reachable by each tuple  $t$  when joining it with other tuples in its subtree. This bottom-up computation is essentially a form of Dynamic Programming.

The algorithm is easier to present using tuple weights instead of attribute weights. We set the weights of  $R$  to  $x_1 + x_2$ , of  $S$  to  $x_3$ , of  $T$  to  $x_4$  and of  $U$  to  $x_5$ . Such a conversion is always possible in linear time, which means that both regimes are supported in the algorithm. We only need to be careful so that the weight of each variable is assigned to a unique relation; this can be achieved through a mapping  $\mu$  that assigns each variable  $x$  in the SUM to the first relation (or atom) that contains  $x$  in the topological sort  $\text{rel}$ . We denote the weight of tuple  $t$  by  $\mathbf{w}(t)$ . Algorithm 3 computes  $\text{opt}(t)$  for all tuples  $t$  by aggregating the input weights using  $\min$  and  $+$ , bottom-up in reverse  $\text{rel}$  order  $\textcircled{4} \rightarrow \textcircled{3} \rightarrow \textcircled{2} \rightarrow \textcircled{1}$ , as shown in Figure 6. The leaf relations set  $\text{opt}(t)$  to be equal to  $\mathbf{w}(t)$ . For tuple  $T(2, 2)$ , which is in the non-leaf relation  $T$ , we add its own weight 2 to the message  $\min\{1, 2\}$  from the joining group in  $U$ , hence  $\text{opt}(T(2, 2)) = 2 + \min\{1, 2\} = 3$ . For a relation with multiple children, we add the messages from all of them. E.g., for  $R(2, 2)$ , we add its own weight 4 with the message  $\min\{3, 5\}$  from  $S$  and the message  $\min\{3\}$  from  $T$ , hence  $\text{opt}(R(2, 2)) = 10$ . By the end of the preprocessing step, we know the optimal weight  $\text{opt}(t)$  for each tuple  $t$ , and the join index entries are *sorted* according to these values.



**Figure 6:** Bottom-up Dynamic Programming for SUM (preprocessing). Orange circles show the weights  $\mathbf{w}(t)$  assigned to tuples. Blue squares show the calculated minimal subtree weight  $\text{opt}(t)$  for each tuple.

**Algorithm 3:** Preprocessing for enumeration by SUM (Section 4.1). Changes compared to Algorithm 1 are in blue.

**Input:** acyclic JQ  $Q$  (without self-joins), database  $D$  with tuple weights, join tree  $\mathcal{T}$ , relation ordering  $\text{rel}$  consistent with  $\mathcal{T}$   
**Output:** reduced and sorted database  $D'$ , JoinIndex $_{R \rightarrow S}$  for each parent  $R$  and child  $S$  in  $\mathcal{T}$   
1 Initialize  $\text{opt}(t) = \mathbf{w}(t)$  for all tuples  $t$  of all relations  
2  $\text{rel} =$  relation ordering consistent with  $\mathcal{T}$   
3 //Process relations in reverse  $\text{rel}$  order (bottom-up in  $\mathcal{T}$ )  
4 for  $i = \ell$  down to 2 do  
5 relation  $S = \text{rel}[i]$ ; relation  $R = \text{pr}(S)$   
6 //Relation  $S$  has been reduced in a previous iteration (or is a leaf)  
7 Construct JoinIndex $_{R \rightarrow S}$  on shared attributes  
8 Sort JoinIndex $_{R \rightarrow S}$  entries by  $\text{opt}$   
9 for tuple  $t \in R$  do  
10  $M = \text{JoinIndex}_{R \rightarrow S}(t)$   
11 //Memoization:  $\text{val}(M)$  is reused  
12 if  $\text{opt}(M)$  not already computed then  
13 |  $\text{opt}(M) = \min\{\infty, \min_{t' \in M} \text{opt}(t')\}$   
14  $\text{opt}(t) = \text{opt}(t) + \text{opt}(M)$   
15 if  $\text{opt}(t) == \infty$  then remove  $t$  from  $R$  in  $D$   
16 Sort the root  $R$  by  $\text{opt}$   
17 return  $D$ , JoinIndex $_{R \rightarrow S}$  for all  $(R, S) \in \mathcal{T}$

REMARK 1. The fact that Algorithm 3 is so similar to the semijoin reduction in Algorithm 1 is not a coincidence. They are both instances of the FAQ framework [2] with different semirings. In particular, the aggregation operators  $\vee$  and  $\wedge$  from the semi-join reduction are replaced with  $\min$  and  $+$  in the variant for SUM. In more technical terms, the former corresponds to the Boolean semiring and the latter to the tropical semiring.

#### 4.2 Top-down Enumeration Phase

As can be seen in Algorithm 4, the high-level logic of the enumeration is the same as the lexicographic enumeration of Algorithm 2. Lines 14 and 25 gen-



**Algorithm 4:** Ranked enumeration for SUM orders. Changes compared to Algorithm 2 are shown in blue.

---

**Input:** acyclic JQ  $Q$ , database  $D$ , SUM order  $\mathcal{W}$   
**Output:** Ranked enumeration of  $Q(D)$  in  $\mathcal{W}$  order

- 1 Remove self-joins from  $Q$  by copying the corresponding relations and renaming them in both  $D$  and  $Q$
- 2 Construct a join tree  $\mathcal{T}$  of  $Q$  with tree-consistent relation order given by  $\text{rel}(i), i \in [\ell]$
- 3 Convert attribute weights to tuple weights
- 4 Preprocess( $Q, D, \mathcal{T}, \text{rel}$ ) (Algorithm 3)
- 5 //A partial answer in the stack is represented as a list of input tuples together with their positions in the corresponding join groups (to easily get the next)
- 6 Initialize **priority queue**  $\mathcal{P}$  with element  $[(t_1, 1)]$  where  $t_1 = R[1]$  and  $R$  is the root of  $\mathcal{T}$
- 7 **repeat**
- 8   //Pop a partial answer ( $1 \leq r \leq |\mathcal{T}|$ ), which also contains the positions  $j_i$  for each tuple  $t_i$
- 9    $s = \mathcal{S}.\text{pop}()$ ;  $[(t_1, j_1), \dots, (t_r, j_r)] = s$
- 10   //Look up matches in  $r$ -th relation
- 11    $M_r = \text{JoinIndex}_{\text{pr}(r) \rightarrow \text{rel}(r)}(t_{\text{pr}(r)})$
- 12   //Push partial answer with next tuple of  $r$ -th relation. It exists if  $j_r$  is not the last position in the group  $M_r$
- 13   **if**  $|M_r| \geq j_r + 1$  **then**
- 14      $s' = s.\text{copy}().\text{replaceLast}((M_r[j_r + 1], j_r + 1))$
- 15      $\mathcal{P}.\text{push}(s')$  with **priority**  $\text{prio}(s')$
- 16   //Range over the remaining relations
- 17   **for**  $i$  from  $r + 1$  to  $|\mathcal{T}|$  **do**
- 18     //Look up matches in  $i$ -th relation
- 19      $M_i = \text{JoinIndex}_{\text{pr}(i) \rightarrow \text{rel}(i)}(t_{\text{pr}(i)})$
- 20     //Extend partial answer with first tuple in matches of  $i$ -th relation
- 21      $t_i = M_i[1]$ ;  $s.\text{append}((t_i, 1))$
- 22     // $s$  is now  $[(t_1, j_1), \dots, (t_r, j_r), \dots, (t_i, 1)]$
- 23     **if**  $|M_i| \geq 2$  **then**
- 24       //Push partial answer with next tuple of  $i$ -th relation to **priority queue**  $\mathcal{P}$
- 25        $s' = s.\text{copy}().\text{replaceLast}((M_i[2], 2))$
- 26        $\mathcal{P}.\text{push}(s')$  with **priority**  $\text{prio}(s')$
- 27     Merge  $s$  into single tuple and output
- 28 **until** query is interrupted or  $\mathcal{P}$  is empty

---

erate new query answers with the tuple in the next position in the join group, like before. However, tuples within a join group are now sorted by  $\text{opt}$ , so each answer generated is guaranteed to produce the next-best weight (among those in the same join group) when extended to a complete answer.

Another important difference is that the stack that maintains the partial answers is replaced by a *priority queue*  $\mathcal{P}$ . Initially,  $\mathcal{P}$  only contains a partial answer with  $R[1] = R(2, 2)$ , producing the top-1 answer  $(2, 2, 3, 2, 1)$  with weight  $\text{opt}(R(2, 2)) = 10$ . The second iteration has 3 candidates in  $\mathcal{P}$ :  $(1, 1)$ ,  $(2, 2, 5)$  and  $(2, 2, 3, 2, 2)$ . The priority of each candidate  $s$ , denoted by  $\text{prio}(s)$  is the weight of the answer we will obtain if we fully extend it. We compute it before inserting it into  $\mathcal{P}$ ; we can either prematurely extend it into a full answer, or we can subtract from the previous answer the weight of the subtree that was removed and add the weight of the new subtree. For example, for  $(2, 2, 5)$ , we can

subtract the weight of the subtree rooted at  $S(2, 3)$  and add the new weight  $\text{opt}(S(2, 5))$  to the weight of the answer of the previous iteration, yielding  $10 - 3 + 5 = 12$ . Based on the priorities,  $(2, 2, 3, 2, 2)$  with priority 11 will be the winner in the second iteration, and the enumeration continues accordingly. Figure 7 depicts the process.

The size of the priority queue  $\mathcal{P}$  is at most  $k\ell$ , since we push at most  $\ell$  candidates in each iteration. Hence, the time of each iteration now includes a logarithmic cost for priority-queue operations (instead of the earlier constant one for stack accesses). However, if we ignore logarithmic factors, the  $\text{TT}(k)$  complexity remains the same as in Theorem 2.

**THEOREM 3 (SUM).** *Let  $Q$  be an acyclic join query over database  $D$  and  $\mathcal{W}$  a SUM ranking function. Ranked enumeration of  $Q(D)$  by  $\mathcal{W}$  can be achieved with  $\text{TT}(k) = \tilde{O}(n + k)$ .*

### 4.3 Performance in Practice

Any- $k$  (enumeration by SUM) has been implemented and the experimental results from PVLDB'20 [44] and PVLDB'21 [47] have been independently reproduced.<sup>8</sup> In Figure 8, we repeat and show an experiment from PVLDB'20 [44] that measures  $\text{TT}(k)$  for a 4-path query (joining relations in a chain) on synthetic data.

The experiment compares ① Any- $k$  against ② JOINFIRST (computing the full result with the Yannakakis algorithm [55]), and ③ PSQL (PostgreSQL 9.5.20).  $\text{TT}(k)$  is depicted on the x-axis and  $k$  on the y-axis. By the time JOINFIRST returns the first answer (in 10.7 sec), any- $k$  has already returned more than 4 million, starting with the first one after 67 msec. PSQL follows an approach similar to JOINFIRST and is outperformed for the top-ranked answers. For the last answer, any- $k$  is slower by less than a factor of 3.

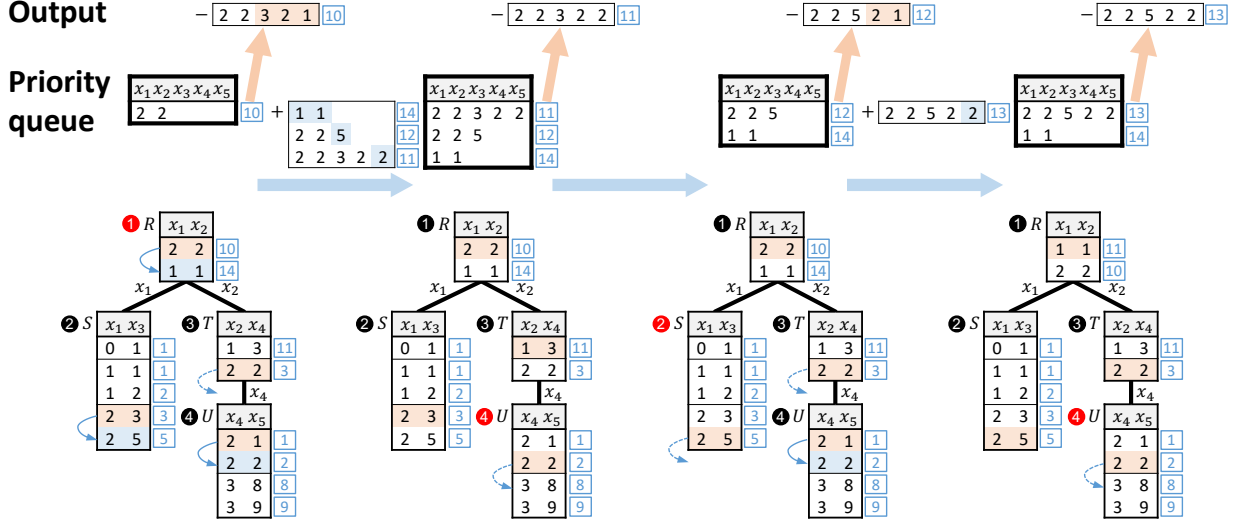
## 5 More General Queries and Tasks

We review a number of generalizations that have been studied, going beyond the task of ranked enumeration by SUM for acyclic JQs.

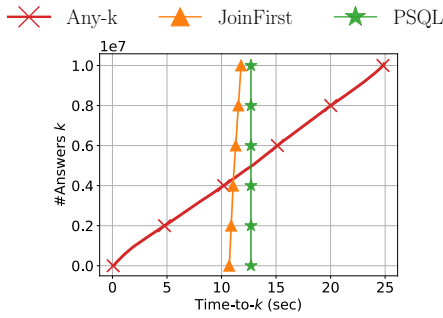
### 5.1 General Ranking Functions

Beyond lexicographic orders and SUM, the algorithm of Section 4 can be used with any ranking function that obeys a property called *subset-monotonicity*. Recall that a ranking function  $w$  maps the query answers to a domain  $W$  ordered by  $\preceq$ . We consider ranking functions that achieve

<sup>8</sup>The code is available to use at <https://github.com/northeastern-datalab/anyk-code>.



**Figure 7:** The example from Fig. 5 adapted with a priority queue instead of a stack to support ranked enumeration by SUM.



**Figure 8:** Any- $k$  against the join-then-rank approach (JOINFIRST) and PostgreSQL (PSQL): Any- $k$  returns the top answer in 67 msec, whereas JOINFIRST needs 10.7 sec [44].

this by aggregating (a multiset of) input weights via an aggregate function  $w_A$ . For example,  $w_A$  is  $\sum$  for SUM.

**DEFINITION 4 (SUBSET-MONOTONICITY).**  
A ranking function  $w$  is subset-monotone if  $w_A(X_1) \preceq w_A(X_2) \Rightarrow w_A(X_1 \uplus Y) \preceq w_A(X_2 \uplus Y)$  for all  $X_1, X_2, Y \in \mathbb{N}^W$ , where  $\uplus$  is multiset union.

Intuitively, subset-monotonicity allows to infer the ranking of complete solutions from the ranking of partial solutions. This is essentially enabling Dynamic Programming [10]. Any subset-monotone ranking function can be handled efficiently with the  $\tilde{O}(n+k)$  guarantee for acyclic queries [43, 46].<sup>9</sup> For

<sup>9</sup>Alternatively, the ranking function can be defined as a selective dioid [44], which can be shown to obey subset-monotonicity.

instance, we may choose the aggregate function to be max instead of  $\sum$ . Under this ranking, only the highest weight is relevant for ordering the answers.

Deep and Koutris [24] generalize subset-monotonicity<sup>10</sup> so that the property is sensitive to the join tree structure; to achieve the desired guarantee, the property needs to hold only across the specific nodes of the join tree used by the algorithm. As an example, consider  $f(x, y) + g(z)$  for arbitrary  $f, g$  and the query  $Q(x, y, z) :- R(x, y), S(y, z)$ . Even though this ranking function is not subset-monotone, it can be supported efficiently because  $x$  and  $y$  are encountered together.

What about other ranking functions? A known negative result is that if the ranking function is a black box, then one cannot do better than materializing the entire query output [24]. Thus, the only guarantee we can hope for is the worst-case output size of the query, given by the AGM bound [5].

## 5.2 CQs with Projection

So far, we have focused on join queries, yet CQs may also contain projection. Projections introduce a new challenge: even if ranked enumeration is efficient for a join query, this may not be true for projections, because we need to eliminate duplicates (under set semantics), potentially increasing  $TT(k)$ .

Bagan et al. [6] established a dichotomy for unranked enumeration that precisely characterizes queries that admit  $TT(k) = \tilde{O}(n+k)$ . The negative side of the dichotomy applies only to self-

<sup>10</sup>Subset-monotonicity is also referred to as a “totally decomposable ranking” [24].

join-free CQs and relies on two complexity-theoretic hypotheses: SPARSEBMM [11] states that two Boolean matrices  $A$  and  $B$ , represented as lists of non-zeros, cannot be multiplied in time  $m^{1+o(1)}$  where  $m$  is the number of non-zeros in  $A$ ,  $B$ , and  $AB$ . HYPERCLIQUE [1, 34] states that for every  $k \geq 2$ , there is no  $O(n \text{ polylog } n)$  algorithm to decide the existence of a  $(k+1, k)$ -hyperclique in a  $k$ -uniform hypergraph with  $n$  hyperedges, where a  $(k+1, k)$ -hyperclique is a set of  $k+1$  vertices such that every subset of  $k$  vertices forms a hyperedge, and a  $k$ -uniform hypergraph is one where all hyperedges contain exactly  $k$  vertices. Under these assumptions, the only efficient (self-join-free) CQs are those that are *free-connex*. A CQ is free-connex if it is acyclic and additionally, it remains acyclic if we add an atom that contains all free variables [15].

Interestingly, that frontier of tractability for unranked enumeration turns out to be the same for ranked enumeration with subset-monotone ranking functions (modulo logarithmic factors).

**THEOREM 5 (DICHOTOMY [43, 46]).** *Let  $Q$  be a CQ. If  $Q$  is free-connex, then ranked enumeration with a subset-monotone ranking function is possible with  $\text{TT}(k) = \tilde{O}(n + k)$ . Otherwise, if it is also self-join-free, then it is not possible with  $\text{TT}(k) = \tilde{O}(n + k)$  for any ranking function, assuming SPARSEBMM and HYPERCLIQUE.*

For the class of acyclic but non-free-connex CQs, the dichotomy precludes the existence of an algorithm with the efficient  $\tilde{O}(n + k)$  guarantee. However,  $\tilde{O}(n \cdot k)$  is possible for subset-monotone ranking functions. This result has been established by the algorithm of Bagan et al. [6] for lexicographic orders, by Deep et al. [23] for lexicographic orders and SUM through a different algorithm, and by Kimelfeld and Sagiv [32] for all subset-monotone ranking functions through a third algorithm.

**THEOREM 6 (NON-FREE-CONNEX [6, 23, 32]).** *Let  $Q$  be an acyclic, non-free-connex CQ. Ranked enumeration of  $Q(D)$  with a subset-monotone ranking function is possible with  $\text{TT}(k) = \tilde{O}(n \cdot k)$ .*

### 5.3 Beyond Acyclic CQs

We can apply the ranked-enumeration algorithms even to queries that are not acyclic CQs, albeit with adjusted complexity guarantees. This is possible if the query can be transformed into an acyclic and free-connex CQ, or a union of such queries. In that case, we first apply the transformation and then perform ranked enumeration on the resulting queries. To deal with a union, we maintain a top-level priority queue that retrieves the next query

answer from the query with the lowest weight in each iteration. Duplicate answers introduce potential complications, but as long as the number of duplicates per answer is bounded by a constant, they can be filtered on-the-fly without increasing complexity. In general, identifying such transformations is an orthogonal research problem, and we discuss three notable cases.

**Cyclic JQs.** For cyclic JQs, we can employ (hyper)tree decompositions [27] to reduce them to a union of acyclic JQs. A decomposition is associated with a width parameter that captures the degree of acyclicity of the query and affects the complexity of subsequent algorithms; for a JQ with width  $d$ , we can achieve  $\text{TT}(k) = \tilde{O}(n^d + k)$ . The state-of-the-art width for a JQ  $Q$  is the submodular width  $\text{subw}(Q)$  [3, 35], transforming a cyclic JQ over a database of size  $n$  to a union of acyclic JQs of size  $\tilde{O}(n^{\text{subw}(Q)})$ , allowing ranked enumeration with  $\text{TT}(k) = \tilde{O}(n^{\text{subw}(Q)} + k)$ .<sup>11</sup>

**Built-in Predicates.** Another case involves acyclic JQs that additionally contain built-in predicates [50] such as inequalities. For *non-equalities* (or “disequalities”  $\neq$ ), we can always achieve  $\text{TT}(k) = \tilde{O}(n + k)$  regardless of where the non-equalities appear in the JQ through a “color-coding” technique [39]. Abo Khamis et al. [31] showed that the same is true for a multidimensional generalization of non-equality, called a Not-All-Equal (NAE) predicate. For *inequalities* ( $<$ ,  $>$ ), we can successfully reduce the query to an acyclic JQ over an  $\tilde{O}(n)$  database, hence achieving  $\text{TT}(k) = \tilde{O}(n + k)$ , as long as the inequality predicate involves variables that appear in join-tree nodes that are adjacent [47]. This condition can be checked directly from the query structure; it is equivalent to the absence of a chordless path of length at least 4 connecting the inequality variables, in the query’s hypergraph [45].

**CQs with FDs.** While a CQ may be acyclic but not free-connex, or even cyclic, it may still be possible to transform it to an acyclic CQ without employing a hypertree decomposition, which generally increase the complexity. This is the case when Functional Dependencies (FDs) are present in the CQ. We can achieve  $\text{TT}(k) = \tilde{O}(n + k)$  for queries whose so-called *FD-extension*, also known as the *closure* of  $Q$  [26], is free-connex [19].

### 5.4 Direct Access

A problem that is closely related to ranked enumeration is *direct access* [21, 22, 25], which asks whether it is possible to efficiently jump to arbitrary

<sup>11</sup>An analog exists for CQs (with projection) [12].

positions in the (implicit) output array, after a preprocessing phase. Ranked enumeration is a special case of this problem, where the accessed positions are  $1, 2, 3, \dots$

Interestingly, the absence of disruptive trios (Definition 1) that describes the feasible lexicographic orders for the algorithm of Section 3 also appears as a necessary condition for achieving direct access with quasilinear preprocessing and polylogarithmic delay [21] (assuming SPARSEBMM). The other necessary condition is for the (self-join-free) acyclic CQ to be  $L$ -connex for the variables  $L$  that appear in the lexicographic order; similarly to the free-connex property, this means that the query remains acyclic when we add a hyperedge consisting of the  $L$  variables. These two conditions are also sufficient for acyclic CQs and thus, provide a dichotomy for self-join-free CQs, under SPARSEBMM and HYPERCLIQUE.

A similar, but much more restrictive on the positive side, dichotomy has been established for SUM [21]. Going beyond quasilinear preprocessing time, Bringman et al. [17] derived precise bounds for each JQ and lexicographic order, Eldar et al. [25] considered queries with aggregation, while Tziavelis et al. [45] studied the problem of a single access where no preprocessing is required.

## 6 Conclusion and Future Outlook

In this paper, we explored the problem of ranked enumeration without fully materializing the query result. We discussed how, for acyclic join queries, certain lexicographic orders can naturally be produced by the unranked enumeration algorithm (with an additional sorting of individual relations). However, not all orders can be handled in this straightforward way. With additional preprocessing and data structures for prioritization, we presented an extended algorithm capable of handling more complex ranking functions, including SUM. Notably, for free-connex CQs, this approach achieves  $TT(k) = \tilde{O}(n+k)$  for any subset-monotone ranking function, and no other self-join-free CQ admits this guarantee (under common hypotheses). Broader classes of queries are also within the reach of the algorithm, as long as they can be efficiently reduced to a union of acyclic and free-connex CQs.

These results are part of an extensive line of research in database theory, focused on the computational tasks that can be efficiently performed on query results without explicitly materializing them. The goal is to offer the illusion of a materialized result, while the actual operations are executed directly on the database. Beyond ranked enumera-

tion and direct access, related tasks include aggregation [2], linear regression [37], and  $k$ -means clustering [36], among others.

One of the areas lacking a refined understanding for ranked enumeration is the complexity landscape for ranking functions. Although some orders are algorithmically easier to achieve than others within the subset-monotone class, their complexity is the same, modulo logarithmic factors. On the other end of the spectrum, for arbitrary black-box ranking functions, no strong guarantees can be achieved. What about the space in-between? To contrast this with the problem of direct access, more intriguing, polynomial-time separations are known even within the class of lexicographic and SUM ranking functions. Mapping out properties of ranking functions and their impact on complexity is an interesting research direction.

Similarly, more work is needed to understand the fundamental difficulty of ranking. For instance, are there surprising cases where ranked enumeration is harder than unranked? One avenue to approach this question is to study CQs with “long” inequalities (in contrast to the “short” inequalities of Section 5.3). For queries, such as  $Q(x_1, x_2, x_3, x_4) :- R(x_1, x_2), S(x_2, x_3), T(x_3, x_4), x_1 < x_4$ , it is known that unranked enumeration can be achieved with  $\tilde{O}(n+k)$  [52], yet ranked enumeration has not been studied. Another avenue is to consider different classes of circuits [4] instead of CQs in order to find such a separation.

The relationship between ranked enumeration and top- $k$  can also lead to interesting questions. Top- $k$  introduces two relaxations, the exact impact of which is not entirely clear: (1)  $k$  is a small constant, and (2)  $k$  is known in advance.

Finally, parallelization is a natural, but challenging, direction. The prioritization of answers required by ranked enumeration implies a degree of sequentiality in the computation, making a parallel adaptation non-obvious. On the theoretical side, the widely used MPC model [8] does not seem to be a good fit because of its batch-processing nature.

**Acknowledgements.** This work was supported in part by a grant from PricewaterhouseCoopers (PwC), the National Institutes of Health (NIH) under award number R01 NS091421, and by the National Science Foundation (NSF) under award numbers CAREER IIS-1762268 and IIS-1956096. Nikolaos Tziavelis was additionally supported by a Google PhD fellowship. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.



## References

- [1] A. Abboud and V. V. Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS*, pages 434–443, 2014. DOI: [10.1109/FOCS.2014.53](https://doi.org/10.1109/FOCS.2014.53).
- [2] M. Abo Khamis, H. Q. Ngo, and A. Rudra. FAQ: questions asked frequently. In *PODS*, pages 13–28, 2016. DOI: [10.1145/2902251.2902280](https://doi.org/10.1145/2902251.2902280).
- [3] M. Abo Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *PODS*, pages 429–444, 2017. DOI: [10.1145/3034786.3056105](https://doi.org/10.1145/3034786.3056105).
- [4] A. Amarilli, P. Bourhis, F. Capelli, and M. Monet. Ranked Enumeration for MSO on Trees via Knowledge Compilation. In *ICDT*, volume 290, 25:1–25:18, 2024. DOI: [10.4230/LIPIcs.ICDT.2024.25](https://doi.org/10.4230/LIPIcs.ICDT.2024.25).
- [5] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013. DOI: [10.1137/110859440](https://doi.org/10.1137/110859440).
- [6] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic (CSL)*, pages 208–222, 2007. DOI: [10.1007/978-3-540-74915-8\\_18](https://doi.org/10.1007/978-3-540-74915-8_18).
- [7] N. Bakibayev, T. Kočíský, D. Olteanu, and J. Závodný. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013. DOI: [10.14778/2556549.2556579](https://doi.org/10.14778/2556549.2556579).
- [8] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6), 2017. DOI: [10.1145/3125644](https://doi.org/10.1145/3125644).
- [9] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, 1983. DOI: [10.1145/2402.322389](https://doi.org/10.1145/2402.322389).
- [10] R. Bellman. The theory of dynamic programming. *Bull. Amer. Math. Soc.*, 60(6):503–515, Nov. 1954. URL: <https://projecteuclid.org:443/euclid.bams/1183519147>.
- [11] C. Berkholz, F. Gerhardt, and N. Schweikardt. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News*, 7(1):4–33, 2020. DOI: [10.1145/3385634.3385636](https://doi.org/10.1145/3385634.3385636).
- [12] C. Berkholz and N. Schweikardt. Constant delay enumeration with FPT-preprocessing for conjunctive queries of bounded submodular width. In *MFCS*, volume 138 of *LIPIcs*, 58:1–58:15, 2019. DOI: [10.4230/LIPIcs.MFCS.2019.58](https://doi.org/10.4230/LIPIcs.MFCS.2019.58).
- [13] P. A. Bernstein and D. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981. DOI: [10.1145/322234.322238](https://doi.org/10.1145/322234.322238).
- [14] M. S. Boddy. Anytime problem solving using dynamic programming. In *AAAI*, pages 738–743, 1991. URL: <https://dl.acm.org/doi/abs/10.5555/1865756.1865791>.
- [15] J. Brault-Baron. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. PhD thesis, Université de Caen, 2013. URL: <https://hal.archives-ouvertes.fr/tel-01081392>.
- [16] J. Brault-Baron. Hypergraph acyclicity revisited. *ACM Comput. Surv.*, 49(3), Dec. 2016. DOI: [10.1145/2983573](https://doi.org/10.1145/2983573).
- [17] K. Bringmann, N. Carmeli, and S. Mengel. Tight fine-grained bounds for direct access on join queries. In *PODS*, 427–436, 2022. DOI: [10.1145/3517804.3526234](https://doi.org/10.1145/3517804.3526234).
- [18] F. Capelli and Y. Strozecki. Geometric Amortization of Enumeration Algorithms. In *STACS 2023*, volume 254, 18:1–18:22, 2023. DOI: [10.4230/LIPIcs.STACS.2023.18](https://doi.org/10.4230/LIPIcs.STACS.2023.18).
- [19] N. Carmeli and M. Kröll. Enumeration complexity of conjunctive queries with functional dependencies. *Theory Comput. Syst.*, 64(5):828–860, 2020. DOI: [10.1007/s00224-019-09937-9](https://doi.org/10.1007/s00224-019-09937-9).
- [20] N. Carmeli and M. Kröll. On the enumeration complexity of unions of conjunctive queries. *TODS*, 46(2), 2021. DOI: [10.1145/3450263](https://doi.org/10.1145/3450263).
- [21] N. Carmeli, N. Tziavelis, W. Gatterbauer, B. Kimelfeld, and M. Riedewald. Tractable orders for direct access to ranked answers of conjunctive queries. *TODS*, 48(1), 2023. DOI: [10.1145/3578517](https://doi.org/10.1145/3578517).
- [22] N. Carmeli, S. Zeevi, C. Berkholz, B. Kimelfeld, and N. Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. In *PODS*, pages 393–409, 2020. DOI: [10.1145/3375395.3387662](https://doi.org/10.1145/3375395.3387662).
- [23] S. Deep, X. Hu, and P. Koutris. Ranked enumeration of join queries with projections. *PVLDB*, 15(5):1024–1037, 2022. DOI: [10.14778/3510397.3510401](https://doi.org/10.14778/3510397.3510401).
- [24] S. Deep and P. Koutris. Ranked enumeration of conjunctive query results. In *ICDT*, volume 186, 5:1–5:19, 2021. DOI: [10.4230/LIPIcs.ICDT.2021.5](https://doi.org/10.4230/LIPIcs.ICDT.2021.5).
- [25] I. Eldar, N. Carmeli, and B. Kimelfeld. Direct Access for Answers to Conjunctive Queries with Aggregation. In *ICDT*, volume 290, 4:1–4:20, 2024. DOI: [10.4230/LIPIcs.ICDT.2024.4](https://doi.org/10.4230/LIPIcs.ICDT.2024.4).
- [26] W. Gatterbauer and D. Suciu. Dissociation and propagation for approximate lifted inference with standard relational database management systems. *The VLDB Journal*, 26(1):5–30, 2017. DOI: [10.1007/s00778-016-0434-5](https://doi.org/10.1007/s00778-016-0434-5).
- [27] G. Gottlob, G. Greco, N. Leone, and F. Scarcello. Hypertree decompositions: questions and answers. In *PODS*, pages 57–74, 2016. DOI: [10.1145/2902251.2902309](https://doi.org/10.1145/2902251.2902309).
- [28] M. Idris, M. Ugarte, S. Vansummeren, H. Voigt, and W. Lehner. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *VLDB J.*, 29:619–653, 2020. DOI: [10.1007/s00778-019-00590-9](https://doi.org/10.1007/s00778-019-00590-9).

- [29] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top- $k$  query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):11, 2008. DOI: [10.1145/1391729.1391730](https://doi.org/10.1145/1391729.1391730).
- [30] V. M. Jiménez and A. Marzal. Computing the  $K$  shortest paths: a new algorithm and an experimental comparison. In *International Workshop on Algorithm Engineering (WAE)*, pages 15–29. Springer, 1999. DOI: [10.1007/3-540-48318-7\\_4](https://doi.org/10.1007/3-540-48318-7_4).
- [31] M. A. Khamis, H. Q. Ngo, D. Olteanu, and D. Suciu. Boolean tensor decomposition for conjunctive queries with negation. In *ICDT*, 21:1–21:19, 2019. DOI: [10.4230/LIPIcs.ICDT.2019.21](https://doi.org/10.4230/LIPIcs.ICDT.2019.21).
- [32] B. Kimelfeld and Y. Sagiv. Incrementally computing ordered answers of acyclic conjunctive queries. In *International Workshop on Next Generation Information Technologies and Systems (NGITS)*, pages 141–152, 2006. DOI: [10.1007/11780991\\_13](https://doi.org/10.1007/11780991_13).
- [33] E. L. Lawler. A procedure for computing the  $k$  best solutions to discrete optimization problems and its application to the shortest path problem. *Management science*, 18(7):401–405, 1972. DOI: [10.1287/mnsc.18.7.401](https://doi.org/10.1287/mnsc.18.7.401).
- [34] A. Lincoln, V. V. Williams, and R. R. Williams. Tight hardness for shortest cycles and paths in sparse graphs. In *SODA*, pages 1236–1252, 2018. DOI: [10.1137/1.9781611975031.80](https://doi.org/10.1137/1.9781611975031.80).
- [35] D. Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *J. ACM*, 60(6):42:1–42:51, 2013. DOI: [10.1145/2535926](https://doi.org/10.1145/2535926).
- [36] B. Moseley, K. Pruhs, A. Samadian, and Y. Wang. Relational Algorithms for  $k$ -Means Clustering. In *ICALP*, volume 198, 97:1–97:21, 2021. DOI: [10.4230/LIPIcs.ICALP.2021.97](https://doi.org/10.4230/LIPIcs.ICALP.2021.97).
- [37] D. Olteanu and M. Schleich. Factorized databases. *SIGMOD Record*, 45(2), 2016. DOI: [10.1145/3003665.3003667](https://doi.org/10.1145/3003665.3003667).
- [38] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *TODS*, 40(1):2, 2015. DOI: [10.1145/2656335](https://doi.org/10.1145/2656335).
- [39] C. H. Papadimitriou and M. Yannakakis. On the complexity of database queries. *Journal of Computer and System Sciences*, 58(3):407–427, 1999. DOI: [10.1006/jcss.1999.1626](https://doi.org/10.1006/jcss.1999.1626).
- [40] R. Paredes and G. Navarro. *Optimal incremental sorting*. In *ALENEX*. 2006, pages 171–182. DOI: [10.1137/1.9781611972863.16](https://doi.org/10.1137/1.9781611972863.16).
- [41] L. Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Record*, 44(1):10–17, 2015. DOI: [10.1145/2783888.2783894](https://doi.org/10.1145/2783888.2783894).
- [42] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984. DOI: [10.1137/0213035](https://doi.org/10.1137/0213035).
- [43] N. Tziavelis. *Efficient Ranked Access to Database Query Answers*. PhD thesis, Northeastern University, 2024. URL: <https://hdl.handle.net/2047/D20668633>.
- [44] N. Tziavelis, D. Ajwani, W. Gatterbauer, M. Riedewald, and X. Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *PVLDB*, 13(9):1582–1597, 2020. DOI: [10.14778/3397230.3397250](https://doi.org/10.14778/3397230.3397250).
- [45] N. Tziavelis, N. Carmeli, W. Gatterbauer, B. Kimelfeld, and M. Riedewald. Efficient computation of quantiles over joins. In *PODS*, pages 303–315, 2023. DOI: [10.1145/3584372.3588670](https://doi.org/10.1145/3584372.3588670).
- [46] N. Tziavelis, W. Gatterbauer, and M. Riedewald. Any- $k$  algorithms for enumerating ranked answers to conjunctive queries. *CoRR*, abs/2205.05649, 2023. URL: <https://arxiv.org/abs/2205.05649>.
- [47] N. Tziavelis, W. Gatterbauer, and M. Riedewald. Beyond equi-joins: ranking, enumeration and factorization. *PVLDB*, 14(11):2599–2612, 2021. DOI: [10.14778/3476249.3476306](https://doi.org/10.14778/3476249.3476306).
- [48] N. Tziavelis, W. Gatterbauer, and M. Riedewald. Optimal join algorithms meet top- $k$ . In *SIGMOD tutorials*, pages 2659–2665, 2020. DOI: [10.1145/3318464.3383132](https://doi.org/10.1145/3318464.3383132). URL: <https://northeastern-datalab.github.io/topk-join-tutorial/>.
- [49] N. Tziavelis, W. Gatterbauer, and M. Riedewald. Toward responsive DBMS: optimal join algorithms, enumeration, factorization, ranking, and dynamic programming. In *ICDE tutorials*, 2022. DOI: [10.1109/ICDE53745.2022.00299](https://doi.org/10.1109/ICDE53745.2022.00299). URL: <https://northeastern-datalab.github.io/responsive-dbms-tutorial/>.
- [50] J. D. Ullman. *Principles of database and knowledge-base systems, Vol. I*. Computer Science Press, Inc., 1988. URL: <https://dl.acm.org/doi/abs/10.5555/42790>.
- [51] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146, 1982. DOI: [10.1145/800070.802186](https://doi.org/10.1145/800070.802186).
- [52] Q. Wang and K. Yi. Conjunctive queries with comparisons. In *SIGMOD*, pages 108–121, 2022. DOI: [10.1145/3514221.3517830](https://doi.org/10.1145/3514221.3517830).
- [53] X. Yang, D. Ajwani, W. Gatterbauer, P. K. Nicholson, M. Riedewald, and A. Sala. Any- $k$ : anytime top- $k$  tree pattern retrieval in labeled graphs. In *WWW*, pages 489–498, 2018. DOI: [10.1145/3178876.3186115](https://doi.org/10.1145/3178876.3186115).
- [54] X. Yang, M. Riedewald, R. Li, and W. Gatterbauer. Any- $k$  algorithms for exploratory analysis with conjunctive queries. In *International Workshop on Exploratory Search in Databases and the Web (ExploreDB)*, pages 1–3, 2018. DOI: [10.1145/3214708.3214711](https://doi.org/10.1145/3214708.3214711).
- [55] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981. URL: <https://dl.acm.org/doi/10.5555/1286831.1286840>.